Code modernization of Polyhedron benchmark suite

Manel Fernández

Intel HPC Software Workshop Series 2016
HPC Code Modernization for Intel® Xeon and Xeon Phi™
February 18th 2016, Barcelona
Approaches for code modernization

“Evolution”

- Requires “tools” expertise
- Potential speedup x2-x20

“Revolution”

- Requires “domain” expertise
- Potential speedup x20-x200

A “Revolution” might not always be feasible (time constraints, etc.)
Polyhedron benchmark suite

Fortran 90 benchmark suite

• 17 self-contained programs adapted from different domains
  • Most of them originally from John Prentice, 1980-2000 (Quetzal)
  • Modified for benchmarking by John Appleyard, 2000-2011 (Polyhedron, now FortranUK)
• Suite was designed for compiler comparison
  • Performance of different compilers on a single node
  • Compute/memory intensive, single threaded

Code modernization of polyhedron benchmark suite

• Goal is to “quickly” parallelize and vectorise (“evolution” approach)
• Case studies for the following benchmarks
  1. linpk: double precision LINPACK benchmark
  2. induct2: self- and mutual-inductance calculation on a system
  3. channel2: A-grid shallow water model for a meridional channel
  4. gas_dyn2: continuity equations solver modelling flow of a gas in 1D
Methodology

Hardware environment
- Intel® Xeon® E5-2650 v2
  - 2 socket
  - 8 cores per socket
  - 2 HTs per core
- Intel® Xeon Phi™ 7120A
  - 61 cores
  - 4 threads per core

Software environment
- Red Hat Enterprise Linux Server 6.6 (Santiago)
- Intel® Fortran Compiler 15.0.2.164 (build 20150121)
  - -O3 -fp-model fast=2 -align array64byte -ipo -xHost/-mmic
Polyhedron/linpk

Case study #1
Polyhedron/linpk benchmark

Double precision LINPACK benchmark

• Features solving a system of linear equation, $Ax = b$

Benchmark in three steps

1. Generates square matrix $N \times N$ ($N = 16K$ elements)
2. Matrix factorization by Gaussian elimination (DGEFA): $O(n^3)$
3. Solves the resulting system of linear equations (DGESL): $O(n^2)$

Total FLOPs is $\frac{2}{3}n^3 + 2n^2$
Main hotspot: DGEFA

Matrix decomposition with partial pivoting by Gaussian elimination
Invokes BLAS-1 routines DAXPY (98%), IDAMAX, DSCAL (all are inlined)
Polyhedron/linpk (DGEFA)

Goal is to fill upper matrix diagonal with zeroes
Sub-matrix values under the diagonal have to be updated on every k-pivot iteration
Loop parallelization

Inner “i” loop properly autovectorized by the compiler.

Middle “j” loop can be parallelized.

Outer “k” loop (diagonal) has dependencies between iterations.

Function is memory bound!
Use LAPACK version

“How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination”, by Jack Dongarra and Piotr Luszczek

LINPACK, late 70s

- DGEFA
  - IDAMAX
  - DSCAL
  - DAXPY

- DGESL
  - DAXPY
  - DDOT

LAPACK, early 90s

- DGESV
  - DGETRF
  - DGETRF
  - DGEF2
  - DLASWP
  - DSWAP
  - DGEMM
  - DLASWP
  - DTRSM

LAPACK version of linear equation solver included in MKL

```
PROGRAM TM = SECNDS(0.0)
CALL MATGEN(a, LDA, N, b, norma)
CALL DGEFA(a, LDA, N, iptv, info)
CALL DGESL(a, LDA, N, iptv, b, 0)
PROGRAM TM = SECNDS(PROGRAM TM)
```
Performance results

Polyhedron/linpk (N=16K)
Polyhedron/linpk summary

Intel® Xeon Phi™ speedup
- vs Intel® Xeon Phi™ serial: 79x
- vs Intel® Xeon® serial: 28x
- vs “best” Intel® Xeon® parallel: 0.64x

Original benchmark ends to be bandwidth bound
- Intel® Xeon: ~40GB/s, Intel® Xeon Phi™: ~80GB/s
- Significant improvements by moving to LAPACK version using MKL
  - Benchmark is turned into compute bound by using blocking DGEMM

Future investigations
- Investigate Xeon Phi inefficiencies
- Investigate MKL automatic offload behavior
Polyhedron/induct2

Case study #2
Polyhedron/induct2 benchmark

Inductance calculations of a system with a thin metal box and internal/external cylindrical coils

- Integrations required are done either analytically or numerically using Gaussian quadrature techniques

Full process as a sequence of several steps

1. Discretizes the box using rectangular grid elements
2. Then inductance calculations for
   a. Self inductance on each element
   b. Mutual inductance between all elements
   c. Mutual inductance between coils
   d. Mutual inductance between coils and each grid element

Generate PSPICE input file for low-frequency electromagnetic modelling of a Res-Q communications system
Main hotspot(s): computing mutual inductance

```fortran
! outer loop is the integration over the circumference of the coil. the next most
! inner loop is a gaussian integration over the height of the coil. the inner
! most loop is the gaussian integration over the quad

112_upper = 0.0_longreal

! do i = 1, 2*m
   theta = pi*real(i,longreal)/real(m,longreal)
   c_vector(1) = r_coil * cos(theta)
   c_vector(2) = r_coil * sin(theta)
! compute current vector for the coil in the global coordinate system
   coil_tmp_vector(1) = -sin(theta)
   coil_tmp_vector(2) = cos(theta)
   coil_tmp_vector(3) = 0.0_longreal
   coil_current_vec(1) = dot_product(rotate_coil(1,:),coil_tmp_vector(:))
   coil_current_vec(2) = dot_product(rotate_coil(2,:),coil_tmp_vector(:))
   coil_current_vec(3) = dot_product(rotate_coil(3,:),coil_tmp_vector(:))
!
   do j = 1, 9
     c_vector(3) = 0.5 * h_coil * 21gauss(j)
! rotate coil vector into the global coordinate system and translate it
     rot_c_vector(1) = dot_product(rotate_coil(1,:),c_vector(:)) + dx
     rot_c_vector(2) = dot_product(rotate_coil(2,:),c_vector(:)) + dy
     rot_c_vector(3) = dot_product(rotate_coil(3,:),c_vector(:)) + dz
!
   do k = 1, 9
     q_vector(1) = 0.5_longreal * a * (x2gauss(k) + 1.0_longreal)
     q_vector(2) = 0.25_longreal * b2 * (y2gauss(k) + 1.0_longreal)
     q_vector(3) = 0.0_longreal
! rotate quad vector into the global coordinate system
     rot_q_vector(1) = dot_product(rotate_quad(1,:),q_vector(:))
     rot_q_vector(2) = dot_product(rotate_quad(2,:),q_vector(:))
     rot_q_vector(3) = dot_product(rotate_quad(3,:),q_vector(:))
!
! compute and add in quadrature term
     numerator = w1gauss(j) * w2gauss(k) * dot_product(coil_current_vec,current_vector)
     denominator = sqrt(dot_product(rot_c_vector-rot_q_vector, rot_c_vector-rot_q_vector))
     112_upper = 112_upper + numerator/denominator
   end do
 end do
```

Inductance between a grid element and a circular/rectangular coil
- 80% execution time in several loops like this one
- "k" loop fully vectorized
- "dot products" completely unrolled
Computing mutual inductance (cont’d)

1. outer loop is the integration over the circumference of the coil. the next most
2. inner loop is a gaussian integration over the height of the coil. the inner
3. most loop is the gaussian integration over the quad

\[ 112_{upper} = 0.0_{longreal} \]

```
! do i = 1, 2*m
   theta = pi*real(i,longreal)/real(m,longreal)
   c_vector(1) = r_coil * cos(theta)
   c_vector(2) = r_coil * sin(theta)
   compute current vector for the coil in the global coordinate system
   coil_tmp_vector(1) = -sin(theta)
   coil_tmp_vector(2) = cos(theta)
   coil_tmp_vector(3) = 0.0_{longreal}
   coil_current_vec(1) = dot_product(rotate_coil(1,:),coil_tmp_vector(:))
   coil_current_vec(2) = dot_product(rotate_coil(2,:),coil_tmp_vector(:))
   coil_current_vec(3) = dot_product(rotate_coil(3,:),coil_tmp_vector(:))
!
! do j = 1, 9
   c_vector(3) = 0.5 * h_coil * z1gauss(j)
   rotate coil vector into the global coordinate system and translate it
   rot_c_vector(1) = dot_product(rotate_coil(1,:),c_vector(:)) + dx
   rot_c_vector(2) = dot_product(rotate_coil(2,:),c_vector(:)) + dy
   rot_c_vector(3) = dot_product(rotate_coil(3,:),c_vector(:)) + dz
!
! do k = 1, 9
   q_vector(1) = 0.5_{longreal} * a * (x2gauss(k) + 1.0_{longreal})
   q_vector(2) = 0.25_{longreal} * b2 * (y2gauss(k) + 1.0_{longreal})
   q_vector(3) = 0.0_{longreal}
   rotate quad vector into the global coordinate system
   rot_q_vector(1) = dot_product(rotate_quad(1,:),q_vector(:))
   rot_q_vector(2) = dot_product(rotate_quad(2,:),q_vector(:))
   rot_q_vector(3) = dot_product(rotate_quad(3,:),q_vector(:))
!
! compute and add in quadrature term
   numerator = w1gauss(j) * w2gauss(k) * dot_product(coil_current_vec,current_vector)
   denominator = sqrt(dot_product(rot_c_vector-rot_q_vector, rot_c_vector-rot_q_vector))
   112_upper = 112_upper + numerator/denominator
end do
```

“m” is number of quadrature points for theta integrals (currently 10)

“j”, “i” loops not good candidates for parallelization (low trip counts)
Explore caller loop

Compute mutual inductance for every grid element
- Number of grid elements about 2M7!

Good candidate for parallelization, good scalability

All iterations are independent
Parallel DO region

Beware of algorithm “thread privates”

Mark reductions properly

Critical sections when needed
Performance results

Polyhedron/induct2

Grid elements/s (higher is better)
Polyhedron/induct2 summary

Intel® Xeon Phi™ speedup
- vs Intel® Xeon Phi™ serial: 61x
- vs Intel® Xeon® serial: 19x
- vs “best” Intel® Xeon® parallel: 1.33x

Benchmark ends to be compute bound
- Almost zero bandwidth, good IPC
- DP div/sqrt might be limiting performance on Intel® Xeon Phi™

Future investigations
- Check/try Intel® Xeon Phi™ reciprocal div/sqrt
- Try to increase IPC on vectorized loops
Polyhedron/channel2

Case study #3
Polyhedron/channel2 benchmark

Simple A-grid shallow water model for a meridional channel
  • CFD code describing e.g., movement of ocean water
  • Original code by John McCalpin (University of Delaware)

Simulation of a grid of $M \times N$ over $n$ time steps of value $\Delta t$, where
  • Height $h$ of the water above equilibrium level
  • Meridional dependence of Coriolis force $f$
  • Fluid velocity $V_{x,y} = u, v$
  • Gravity $g$
  • $M = 8000$
  • $N = 256$
  • $n = 6580$
Polyhedron/channel2

Input set representing water flood over a terrain
Color gradient to represent height of the water
  • Terrain is blue (height = 0)
  • Other colors to represent height of the water between surface and sea bottom
Identifying hotspot(s)

- ~100% of execution time
- Loops properly vectorized
  - Nice use of array notation
- Some loops nicely fused
  - Improves data locality
- Overall, compiler does a good job with vectorization
Threading with OpenMP parallel loop

Undo array notation in outer dimension

• Keep 2:N-1 iterations within the DO loop
• Hoist 1 and N iterations out of the DO loop

```fortran
!------------------------- BEGIN TIME MARCHING LOOP -----
do istep=1,NSTEPS
  time = istep*dt
  ! ----- interior calculations ------ !
  dudx = ddx(u(:,:,mid))
dvdy = ddy(v(:,:,mid))
dhdx = ddx(h(:,:,mid))
dhdy = ddy(h(:,:,mid))
  u(2:M-1,1,new) = u(2:M-1,1,old) &
                  +2*dt*f(2:M-1,1:N)*v(2:M-1,1,N,mid) &
                  -2*dt/(2*dt)*g*dhdx(2:M-1,1)
  u(2:M-1,N,new) = u(2:M-1,N,old) &
                  +2*dt*f(2:M-1,N)*v(2:M-1,N,mid) &
                  -2*dt/(2*dt)*g*dhdx(2:M-1,N)
  !$OMP PARALLEL DO
  do idx=2,N-1
    !-------------------------
    v(1:M,2:N-1,new) = v(1:M,2:N-1,old) &
                    +2*dt*f(1:M,2:N-1)*u(1:M,2:N-1,mid) &
                    -2*dt/(2*dt)*g*dhdx(1:M,2:N-1)
    h(1:M,2:N-1,new) = h(1:M,2:N-1,old) &
                    +2*dt*f(1:M,2:N-1)*u(1:M,2:N-1,mid) &
                    -2*dt/(2*dt)*g*dhdx(1:M,2:N-1)
    !-------------------------
  end do
  !$OMP END PARALLEL DO
  !-------------------------
  ! ----- boundary calculations ------ !
```

```fortran
  !-------------------------
  ! ----- boundary calculations ------ !
```
Performance results: OpenMP DO loop

Steps per second (higher is better)

- Host, 1t/c
- Host, 2t/c
- MIC, 1t/c
- MIC, 2t/c
- MIC, 3t/c
- MIC, 4t/c

Base
OpenMP DO loop
ddx/ddy contains the stencil part of the algorithm.

Hotspots dominated by inner stencil (unit stride vectorization)
Boundaries also vectorized
  • ddx by using gather/scatter!
Manual inlining & restructuring
Performance results: manual inlining
OpenMP parallel region

- Open parallel region only once
- Restrict critical regions to be executed by master thread only

```c
!--------------------- BEGIN TIME MARCHING LOOP -----
do istep=1,NSTEPS
  time = istep*dt
  ! ------ interior calculations ------ !
  ! ...

!$OMP PARALLEL DO
  do jdx=2,N-1
    ! ...
  end do
!$OMP END PARALLEL DO
  ! ------ boundary calculations ------ !
  ! ...
end do

!--------------------- END TIME MARCHING LOOP -----
```
Performance results: OpenMP parallel region

Steps per second (higher is better)

- Host, 1t/c
- Host, 2t/c
- MIC, 1t/c
- MIC, 2t/c
- MIC, 3t/c
- MIC, 4t/c

Base | OpenMP DO loop | Manual inlining | OpenMP parallel region
Polyhedron/channel2 summary

Intel® Xeon Phi™ speedup

- vs Intel® Xeon Phi™ serial: 44.3x
- vs Intel® Xeon® serial: 13.7x
- vs “best” Intel® Xeon® parallel: 2.6x

Benchmark ends to be bandwidth bound

- Intel® Xeon: ~40GB/s, Intel® Xeon Phi™: ~90GB/s
- Parallel region already bounded by bandwidth limit
- More threads do not help (1t/c already saturates bandwidth)

Future investigations

- Swap inner/outer dimensions <M,N> to <N,M>
  - N dimension not big enough to scale properly among threads
- Investigate possible load imbalance issues
Polyhedron/gas_dyn2

Case study #4
Polyhedron/gas_dyn2 benchmark

Continuity equations solver to model the flow of a gas in 1D
- The expanding gas front is moved by free molecular flow
  - 3M particles, 20K steps
  - Nonlinear PDEs for conservation of mass, momentum and energy

Time-step based main loop
1. Equations of state (EOS) for current configuration
2. Check/fix boundary conditions
3. Choose next time step (CHOZDT)
Polyhedron/gas_dyn2

Temperature, density, and speed of every gas particle within the pipe
Serial version

Two main hotspots: EOS (66%) and CHOZDT(33%)

- Implicit loops by using Fortran 90 array notation
- Both hotspots perfectly fused + vectorized
- 1D iteration space

```fortran
SUBROUTINE CHOZDT(NODES, VEL, SOUND, DX, DT)
  !---------------------------------------------------------------
  INTEGER :: NODES, ISET(1)
  REAL, DIMENSION(NODES) :: VEL, SOUND, DX, DTEMP
  !---------------------------------------------------------------
  DTEMP = DX/(ABS(VEL) + SOUND)
  ISET = MINLOC (DTEMP)
SUBROUTINE EOS(NODES, IENER, DENS, PRES, TEMP, AGAMMA, CS, SHEAT, CGAMMA)
  !---------------------------------------------------------------
  INTEGER NODES
  REAL SHEAT, CGAMMA
  REAL, DIMENSION(NODES) :: IENER, DENS, PRES, TEMP, AGAMMA, CS
  !---------------------------------------------------------------
  TEMP(NODES) = IENER(NODES)/SHEAT
  PRES(NODES) = (CGAMMA - 1.0) * DENS(NODES) * IENER(NODES)
  GAMMA(NODES) = CGAMMA
  CS(NODES) = SQRT(CGAMMA*PRES(NODES)/DENS(NODES))
```
OpenMP workshare construct

Native parallelization with WORKSHARE construct

• Workshare construct currently not working (not parallelized)
• Reduction loop in CHOZDT does not even vectorize
• No clauses for private variables, schedule policy, etc.

```c
!$OMP PARALLEL WORKSHARE
    DTEMP = DX/(ABS(VEL) + SOUND)
    ISET = MINLOC(DTEMP)
!$OMP END PARALLEL WORKSHARE

!$OMP PARALLEL WORKSHARE
    TEMP(:NODES) = IENER(:NODES)/SHEAT
    PRES(:NODES) = (CGAMMA - 1.0)*DENS(:NODES)*IENER(:NODES)
    GAMMA(:NODES) = CGAMMA
    CS(:NODES) = SQRT(CGAMMA*PRES(:NODES)/DENS(:NODES))
!$OMP END PARALLEL WORKSHARE
```
OpenMP parallel loop (EOS)

Straightforward transformation
Loop both parallelized and vectorized

```
!$omp parallel do simd
  do n = 1, nodes
    temp(n) = iener(n)/sheat
    pres(n) = (cgamma - 1.0)*dens(n)*iener(n)
    agamma(n) = cgamma
    cs(n) = sqrt(cgamma*pres(n)/dens(n))
  end do
!$omp end parallel do simd
```
OpenMP parallel loop (CHOZDT)
Horizontal reduction with OpenMP critical section

Intel compiler 15.0 does not support OpenMP 4.0 user defined reductions

We have to write the parallel reduction by ourselves!

*Idea*: local reduction for every thread, then *global reduction* as *CRITICAL* section

*NOWAIT* to alleviate possible imbalances

```fortran
INTEGER :: N, ISET_L
REAL :: ISET_V, ISET_1, DTEMP

!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
! global values for minloc result, local values for every thread
ISET_L = HUGB(ISET_L)
ISET(1) = 0

!$OMP PARALLEL PRIVATE(ISET_V,ISET_L,DTEMP)
  ISET_V = ISET_1
  ISET_L = 1
! compute DTEMP in parallel, also MINLOC for every thread (if)
!$OMP DO SIMD
  DO N = 1, NODES
    DTEMP = DX(N)/(ABS(VEL(N)) + SOUND(N))
    IF (DTEMP < ISET_V) THEN
      ISET_V = DTEMP
      ISET_L = N
    ENDIF
  END DO
!$OMP END DO SIMD NOWAIT

! horizontal reduction (one thread at a time, as critical section)
!$OMP CRITICAL
  IF (ISET_V < ISET_1) THEN
    ISET_1 = ISET_V
    ISET(1) = ISET_L
  ENDIF
!$OMP END CRITICAL
!$OMP END PARALLEL
```
Performance results: critical section

Steps per second (higher is better)

- Host, 1t/c
- Host, 2t/c
- MIC, 1t/c
- MIC, 2t/c
- MIC, 3t/c
- MIC, 4t/c
OpenMP critical not so good as expected

Critical section might be good only when load unbalancing
Getting worse as the number of threads increases
OpenMP parallel loop (CHOZDT)

Horizontal reduction done by master thread

**Idea:** local reduction is consolidated in a **global array** (as many entries as number of threads in parallel region)

**Horizontal reduction** done by master thread out of the parallel region

False sharing occurs, but it causes minimal performance impact

```fortran
INTEGER :: TID
REAL, DIMENSION(NTHREADS) :: ASET_V
INTEGER, DIMENSION(NTHREADS) :: ASET_L

! global values for minloc result, local values for every thread
ISET_1 = HUGE(ISET_1)
ISET(1) = 0

$OMP PARALLEL PRIVATE(ISET_V,ISET_L,DTEMP,TID)
  ISET_V = ISET_1
  ISET_L = 1
! compute DTEMP in parallel, also MINLOC for every thread (if)
$OMP DO SIMD
  DO N = 1, NODES
    DTEMP = DX(N)/(ABS(VEL(N)) + SOUND(N))
    IF (DTEMP < ISET_V) THEN
      ISET_V = DTEMP
      ISET_L = N
    ENDIF
  END DO
$OMP END DO SIMD NOWAIT
! move local data to global array
  TID = OMP_GET_THREAD_NUM() + 1
  ASET_V(TID) = ISET_V
  ASET_L(TID) = ISET_L

$OMP END PARALLEL
! now master thread performs horizontal reduction
!
!DIR$ VECTOR ALWAYS
  ISET = MINLOC(ASET_V)
  ISET_1 = ASET_V(ISET(1))
  ISET(1) = ASET_L(ISET(1))
```
Performance results: serial reduction

![Graph showing performance results for different configurations.](image-url)
Improving vectorization: data alignment

Data must be aligned and compiler must know
- `align array 64 byte`
- `!dir$ vector aligned`

**Problem:** OpenMP 4.0 does not guarantee that loop is split at “vector” boundaries

**Solution:** separate tail to support parallelization + vectorization + alignment
- Not using `SINGLE` region because `VECTOR` pragma is not allowed (!)
Improving vectorization: streaming stores

Use “non temporal” streaming stores (EOS only)

• Some arrays are just written, we don’t need to read them previously

Saves bandwidth for other actual read accesses

```c
!$OMP PARALLEL DO SIMD
!DIRS VECTOR_ALIGNED, NONTEMPORAL(TEMP,PRES,GAMMA,CS)
  DO N = 1, NLOW
    TEMP(N) = IENER(N)/SHEAT
    PRES(N) = (CGAMMA - 1.0)*DENS(N)*IENER(N)
    GAMMA(N) = CGAMMA
    CS(N) = SQRT(CGAMMA*PRES(N)/DENS(N))
  END DO
!$OMP END PARALLEL DO SIMD
! do the tail...
!DIRS VECTOR_ALIGNED, NONTEMPORAL(TEMP,PRES,GAMMA,CS)
  DO N = NLOW+1, NODES
    TEMP(N) = IENER(N)/SHEAT
    PRES(N) = (CGAMMA - 1.0)*DENS(N)*IENER(N)
    GAMMA(N) = CGAMMA
    CS(N) = SQRT(CGAMMA*PRES(N)/DENS(N))
  END DO
```
Performance results: alignment and streaming

Steps per second (higher is better)

- Base
- Critical
- Reduction
- Align/Stream

Host, 1t/c
Host, 2t/c
MIC, 1t/c
MIC, 2t/c
MIC, 3t/c
MIC, 4t/c
Performance results (cont’d): Amdahl’s law

- EOS
- CHOZDT
- Other (serial)

Time (log), seconds (lower is better)
Polyhedron/gas_dyn2 summary

Intel® Xeon Phi™ speedup
- vs Intel® Xeon Phi™ serial: 17.8x
- vs Intel® Xeon® serial: 7.4x
- vs “best” Intel® Xeon® parallel: 1.8x

Benchmark ends to be bandwidth bound
- Intel® Xeon: ~40GB/s, Intel® Xeon Phi™: ~80GB/s
- Parallel parts cannot improve, already bounded by bandwidth limit
- More threads do not help, they tend to increase the serial parts

Future investigations
- Investigate “offload” model
  - Data persistent on Intel® Xeon Phi™ side, move back only boundaries on every step
- Better understanding of bandwidth and parallel/serial limits
General observations

Code modernization of Polyhedron benchmark suite
Polyhedron benchmarks: summary

Main effort in code modernization is about threading/TLP
  • There’s no good “auto-parallelizer” that we can rely on

Compiler does already a good job about exploiting SIMD (DLP)
  • True... at least for Fortran codes!
    • More limitations in C/C++ due to pointer aliasing
    • Explicit/guided vectorization techniques are easily applicable

Reductions are common
  • Always try to exploit data locality by computing local reductions
    • Avoid GPU-style reductions!

Most applications are bandwidth bound
  • Makes hard to take advantage of all (hyper-)threads in a core
  • Try to improve data locality by changing the way how data is being accessed